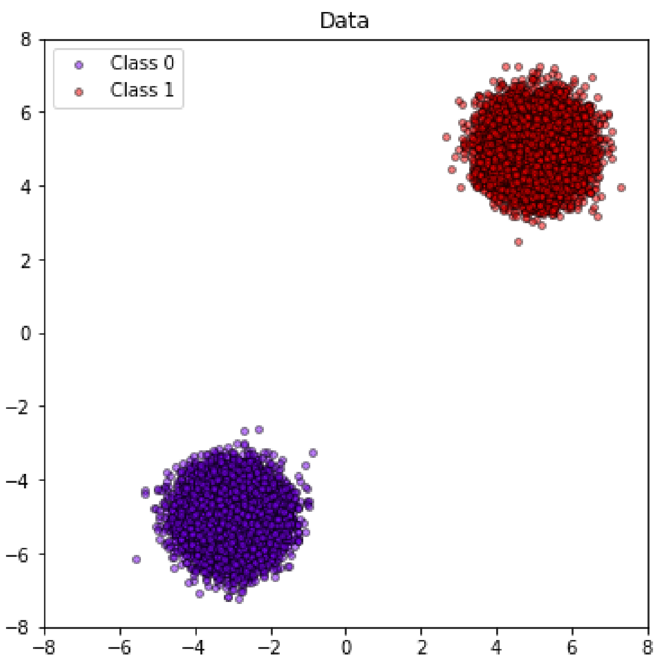
In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

from sklearn.datasets import make_blobs
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import brier_score_loss
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split
```

Type *Markdown* and LaTeX: α^2

In [2]:
```python
n_samples = 50000
n_bins = 3  # use 3 bins for calibration_curve as we have 3 clusters here

# Generate 3 blobs with 2 classes where the second blob contains
# half positive samples and half negative samples. Probability in this
# blob is therefore 0.5.
centers = [(-3, -5), (5, 5)] # not symmetric
X, y = make_blobs(n_samples=n_samples, centers=centers, cluster_std=0.6, shuffle=False,

y[:n_samples // 2] = 0
y[n_samples // 2:] = 1

# split train, test for calibration
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42
plt.figure(figsize=(6, 6))
y_unique = np.unique(y)
colors = cm.rainbow(np.linspace(0.0, 1.0, y_unique.size))
for this_y, color in zip(y_unique, colors):
    this_X = X_train[y_train == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=15,
                c=color[np.newaxis, :],
                alpha=0.5, edgecolor='k',
                label="Class %s" % this_y)
plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.legend(loc="best")
plt.title("Data");
```

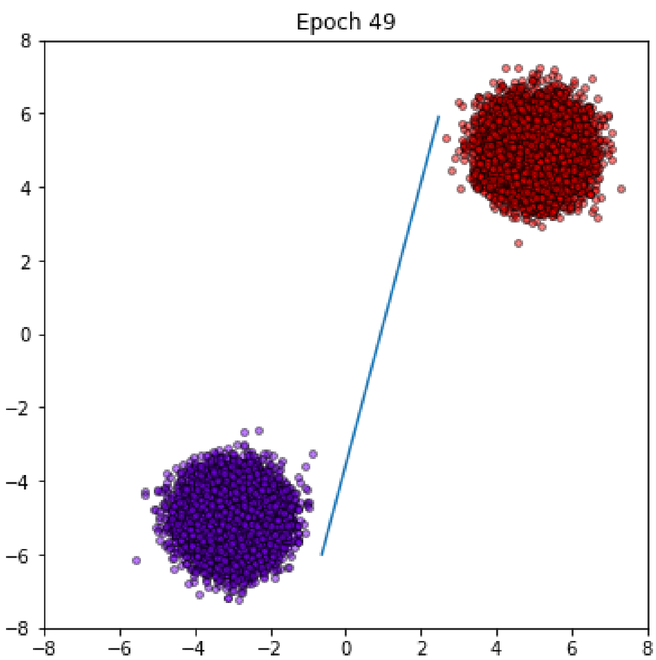


In [3]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributions as D
import torch.nn.functional as F
```

In [4]:
```
class LogisticRegression(nn.Module):

    def __init__(self, in_features, out_features, bias):
        super(LogisticRegression, self).__init__()
        self.forward_block = nn.Sequential(
            nn.Linear(in_features, out_features, bias),
            nn.LogSoftmax()
        )

    def forward(self, x):
        """ calculate log probability
        """
        log_prob = self.forward_block(x)
        return log_prob

    def get_decision_boundary(self):

        """ return a function for the decision boundary (a line on 2D plane)
        """
        # only suitable for 2D toy data
        lin_layer = self.forward_block[0]
        assert lin_layer.in_features==2 and lin_layer.out_features==2

        # equation y = Ax + b, and y_0 = y_1, solve for x_0 = f(x_1)
        A = lin_layer.weight.data
        b = lin_layer.bias.data
        c = A[0,:] - A[1,:]
        d = b[0] - b[1]
        # then the decision boundary is c[0] * x_0 + c[1] * x_1 + d = 0, i.e., np.dot(c,
        # general solution for higher dimension data could be found by scipy.linalg.null
        # but hard to visualize anyway

        return lambda x_1: -1.0 * (d+c[1]*x_1) / c[0]
```

In [5]:
```
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dty
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test, dtype=t
```

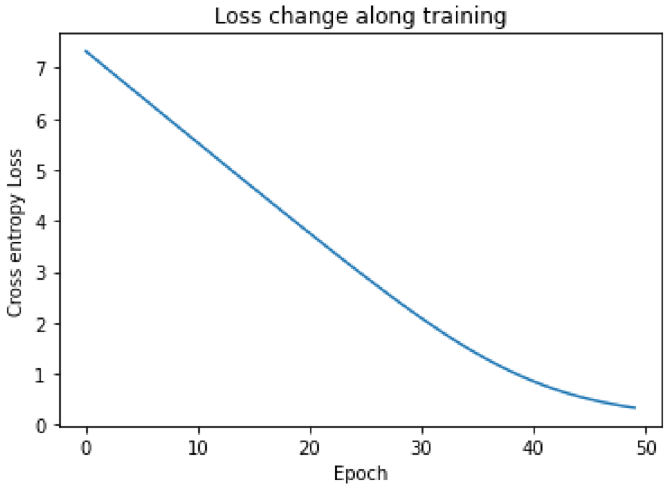
In [6]:
```python
from IPython import display

# linear + softmax = logistic regression
model = LogisticRegression(in_features=2, out_features=2, bias=True)
creterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
loss_list = []

y_unique = np.unique(y)
colors = cm.rainbow(np.linspace(0.0, 1.0, y_unique.size))

fig, ax = plt.subplots(1,figsize=(6,6))

for epoch in range(50):
    pred = model(X_train)
    loss = creterion(pred, y_train)
    loss_list.append(loss.item())
    # print(sum(loss_list)/len(loss_list))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ax.clear()
    for this_y, color in zip(y_unique, colors):
        this_X = X_train[y_train == this_y]
        ax.scatter(this_X[:, 0], this_X[:, 1], s=15,
                   c=color[np.newaxis, :],
                   alpha=0.5, edgecolor='k',
                   label="Class %s" % this_y)

    fn = model.get_decision_boundary()
    x1_s = torch.arange(-6, 6, 0.1)
    x0_s = fn(x1_s)
    ax.plot(x0_s, x1_s)
    ax.set_xlim(-8,8)
    ax.set_ylim(-8,8)
    ax.set_title("Epoch {}".format(epoch))

    display.clear_output(wait=True)
    display.display(fig)
    plt.pause(0.005)
```



In [7]:
```
# verify the decision boundary is correct (two classes share equal probabilities)
xs = torch.cat((x0_s.reshape(-1,1), x1_s.reshape(-1,1)),1).type(torch.float)
log_prob = model(xs)
torch.allclose(log_prob.exp(), torch.tensor(0.5))
```

Out[7]:
```
True
```

In [8]:
```
plt.plot(loss_list)
plt.xlabel("Epoch")
plt.ylabel("Cross entropy Loss")
plt.title("Loss change along training");
```



In [9]:
```
def classification_accuracy(prob, y):
    """ compute classification accuracy given the probability of each class
    """
    value, idx = torch.max(prob, dim=1)
    acc = torch.sum(idx == y)/y.numel()
    return acc

# model outputs log softmax result, so use exp to get back probability
prob = model(X_test).exp()
acc = classification_accuracy(prob, y_test)
print("Classification Accuracy is {}".format(acc))
```

```
Classification Accuracy is 1.0
```

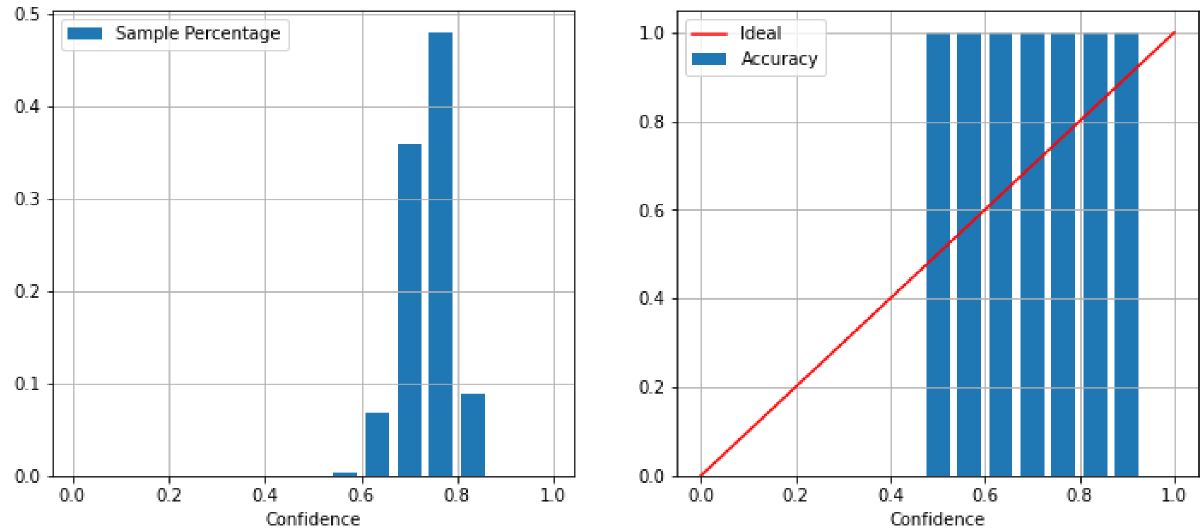
In [10]:
```
class ECELoss(nn.Module):
    """
    Ported from https://github.com/gpleiss/temperature_scaling/blob/master/temperature_s

    Calculates the Expected Calibration Error of a model.
    (This isn't necessary for temperature scaling, just a cool metric).
    The input to this loss is the logits of a model, NOT the softmax scores.
    This divides the confidence outputs into equally-sized interval bins.
    In each bin, we compute the confidence gap:
    bin_gap = | avg_confidence_in_bin - accuracy_in_bin |
    We then return a weighted average of the gaps, based on the number
    of samples in each bin
    See: Naeini, Mahdi Pakdaman, Gregory F. Cooper, and Milos Hauskrecht.
    "Obtaining Well Calibrated Probabilities Using Bayesian Binning." AAAI.
    2015.
    """

    def __init__(self, n_bins=15):
        """
        n_bins (int): number of confidence interval bins
        """
        super(ECELoss, self).__init__()
        bin_boundaries = torch.linspace(0, 1, n_bins + 1)
        self.bin_lowers = bin_boundaries[:-1]
        self.bin_uppers = bin_boundaries[1:]
        # # accuracy, confidence, sample percentage for each bin
        self.acc_list = []
        self.conf_list = []
        self.perc_samples_list = []

    def forward(self, probs, labels, is_logit=False):
        """ compute the expected calibration error of a classification output

        Args:
            probs ([torch.Tensor]): the probability of each class (after softmax), typic
            labels ([torch.Tensor]):  the true class label, typically shape (n_samples,
            is_logit (bool, optional): the input to `probs` will be regarded as logits (
        """
        if is_logit:
            probs = F.softmax(probs, dim=-1)

        confidences, predictions = torch.max(probs, 1)
        accuracies = predictions.eq(labels)

        ece = torch.zeros(1, device=probs.device)
        for bin_lower, bin_upper in zip(self.bin_lowers, self.bin_uppers):
            # Calculated |confidence - accuracy| in each bin
            in_bin = confidences.gt(bin_lower.item()) * confidences.le(bin_upper.item())
            prop_in_bin = in_bin.float().mean()

            if prop_in_bin.item() > 0:
                accuracy_in_bin = accuracies[in_bin].float().mean()
                avg_confidence_in_bin = confidences[in_bin].mean()
                ece += torch.abs(avg_confidence_in_bin - accuracy_in_bin) * prop_in_bin
                self.acc_list.append(accuracy_in_bin.item())
                self.conf_list.append(avg_confidence_in_bin.item())
                self.perc_samples_list.append(prop_in_bin.item())
            else:
                self.acc_list.append(0.0)
                self.conf_list.append(0.0)
                self.perc_samples_list.append(0.0)

        return ece

    def plot_acc_conf_gap(self):
        """ plot the confidence-accuracy gap
            must run a forward pass first
```

```
        """
        if len(self.acc_list) == 0 or len(self.conf_list) == 0:
            print("run forward pass before plotting ece")
            return

        xs = (self.bin_uppers + self.bin_lowers) / 2
        plt.figure(figsize=(12,5))

        plt.subplot(1,2,1)
        plt.grid(True)
        width = 0.05
        plt.bar(xs, np.array(self.perc_samples_list), width, label="Sample Percentage")
        plt.xlabel("Confidence")
        plt.legend()

        plt.subplot(1,2,2)
        plt.grid(True)
        plt.bar(xs, np.array(self.acc_list), width, label="Accuracy")
        plt.plot(np.arange(0,1.1,0.1), np.arange(0,1.1,0.1), label="Ideal", color="r")
        plt.xlabel("Confidence")
        plt.legend()
        plt.show()
```

In [11]:
```
ece_loss = ECELoss(n_bins=15)
# the model outputs log softmax result, so use exp to get back probability
ece_score = ece_loss(model(X_test).exp(), y_test, is_logit=False)
print(ece_score)
```

```
tensor([0.2612], grad_fn=<AddBackward0>)
```

In [12]:
```
ece_loss.plot_acc_conf_gap()
```



In [13]:
```
sto_output = torch.randn(2,5,3,4) # 5 is the number of samples for each deterministic in
true_label = torch.randint(4, (2,1,3))

dist = D.Categorical(logits=sto_output)
dist.log_prob(true_label).shape
```

Out[13]:
```
torch.Size([2, 5, 3])
```

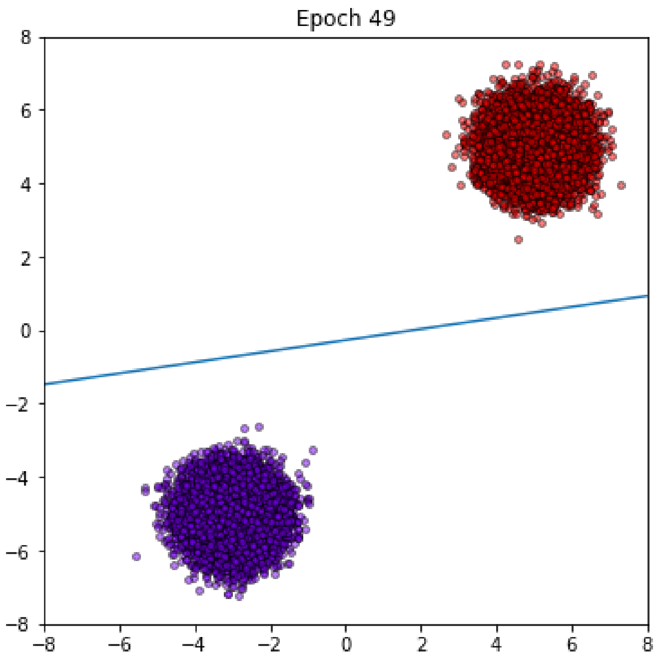
In [14]:
```python
class StoLayer(nn.Module):

    def __init__(self):
        super(StoLayer, self).__init__()
        self.det_layer = None # base on a deterministic layer
        self.base_dist = None # a distribution to draw samples
        self.norm_flow = None # a flow to transform the samples


    def forward(self, x):

        mult_noise = self.base_dist.sample(x)
        transformed_noise, log_det_jacobian = self.norm_flow(mult_noise)
        out = self.det_layer(x*transformed_noise)

        return out, log_det_jacobian
```

In [15]:
```
class NF_Block(nn.Module):

    def __init__(self, depth):
        super(NF_Block, self).__init__()
        self.forward_block = nn.Sequential(AffineTransform(learnable=True),
                                           *[PlanarFlow() for _ in range(depth)])

    def forward(self, samples):

        transformed_samples, log_det_jacobian = self.forward_block(samples)

        return transformed_samples, log_det_jacobian

class PlanarFlow(nn.Module):
    """ modified based on https://github.com/kamenbliznashki/normalizing_flows/blob/mast
    """
    def __init__(self, init_sigma=0.01):
        super(PlanarFlow, self).__init__()
        self.u = nn.Parameter(torch.randn(1, 2).normal_(0, init_sigma))
        self.w = nn.Parameter(torch.randn(1, 2).normal_(0, init_sigma))
        self.b = nn.Parameter(torch.randn(1).fill_(0))

    def forward(self, x, normalize_u=True):
        # allow for a single forward pass over all the transforms in the flows with a Se
        if isinstance(x, tuple):
            z, sum_log_abs_det_jacobians = x
        else:
            z, sum_log_abs_det_jacobians = x, 0

        # normalize u s.t. w @ u >= -1; sufficient condition for invertibility
        u_hat = self.u
        if normalize_u:
            wtu = (self.w @ self.u.t()).squeeze()
            m_wtu = - 1 + torch.log1p(wtu.exp())
            u_hat = self.u + (m_wtu - wtu) * self.w / (self.w @ self.w.t())

        # compute transform
        f_z = z + u_hat * torch.tanh(z @ self.w.t() + self.b)
        # compute log_abs_det_jacobian
        psi = (1 - torch.tanh(z @ self.w.t() + self.b)**2) @ self.w
        det = 1 + psi @ u_hat.t()
        log_abs_det_jacobian = torch.log(torch.abs(det) + 1e-6).squeeze()
        sum_log_abs_det_jacobians = sum_log_abs_det_jacobians + log_abs_det_jacobian

        return f_z, sum_log_abs_det_jacobians

class AffineTransform(nn.Module):
    """ will keep the input unchanged if not learnable
    """
    def __init__(self, learnable=False):
        super().__init__()
        self.mu = nn.Parameter(torch.zeros(2)).requires_grad_(learnable)
        self.logsigma = nn.Parameter(torch.zeros(2)).requires_grad_(learnable)

    def forward(self, x):
        z = self.mu + self.logsigma.exp() * x
        sum_log_abs_det_jacobians = self.logsigma.sum()
        return z, sum_log_abs_det_jacobians
```

In [16]:
```python
class StoLogisticRegression(nn.Module):

    DET_MODEL = LogisticRegression # corresponding class of deterministic model

    def __init__(self, in_features, out_features, bias, mult=True, depth=8):
        """ mult: use multiplicative noise (use additive when set to False)
            got surprisingly bad result when I use multiplicative noise (just for this t
        """
        super(StoLogisticRegression, self).__init__()
        self.lin_layer = nn.Linear(in_features, out_features, bias)
        self.out_layer = nn.LogSoftmax()
        # assume the prior distribution is also this one
        # is this correct ?
        self.base_dist = D.Normal(1.0, 0.1)
        self.norm_flow = NF_Block(depth=depth)
        self.mult = mult

    def forward(self, x):
        """ calculate log probability of each class, and the log_det_jacobian for the st
        """
        samples = self.base_dist.sample(x.shape)
        transformed_samples, log_det_jacobian = self.norm_flow(samples)
        if self.mult:
            x = x*transformed_samples
        else:
            x = x + transformed_samples
        log_probs = self.out_layer(self.lin_layer(x))
        # batched class probability and batched log_det_jacobian
        return log_probs, log_det_jacobian

    def make_prediction(self, x, n_samples=128, return_all_probs=False):
        # draw `n_samples` stochastic noise for each input and treat them as batches
        x = x.repeat_interleave(n_samples, dim=0)
        log_probs, _ = self.forward(x)
        log_probs = log_probs.reshape(-1, n_samples, x.size(1)) # result size (batch_siz
        probs = log_probs.exp()
        # average over the samples, result size (batch_size, n_classes)
        variances, mean_prob = torch.var_mean(probs, dim=1, unbiased=False)
        if return_all_probs:
            return mean_prob, variances, probs
        else:
            return mean_prob, variances

    def kl_div(self, log_det_jacobian):
        # the mean kl over the data points makes an estimation of the true kl
        return - log_det_jacobian.mean()

    def calc_loss(self, log_probs, label, log_det_jacobian):
        # elbo =  log_likelihood - kl_divergence (both averaged over samples)
        # log_likelihood should be averaged over samples and summed over data points
        log_likelihood = D.Categorical(logits=log_probs).log_prob(label).mean()
        # so divide kl again by the number of data points (although already divided by i
        kl_divergence = self.kl_div(log_det_jacobian) / label.size(0)
        # minimize negative elbo
        return - log_likelihood + kl_divergence, log_likelihood, kl_divergence

    def get_decision_boundary(self):

        """ return a function for the decision boundary (a line on 2D plane)
        """
        # only suitable for 2D toy data
        lin_layer = self.lin_layer
        assert lin_layer.in_features==2 and lin_layer.out_features==2

        # equation y = Ax + b, and y_0 = y_1, solve for x_0 = f(x_1)
        A = lin_layer.weight.data
        b = lin_layer.bias.data
```

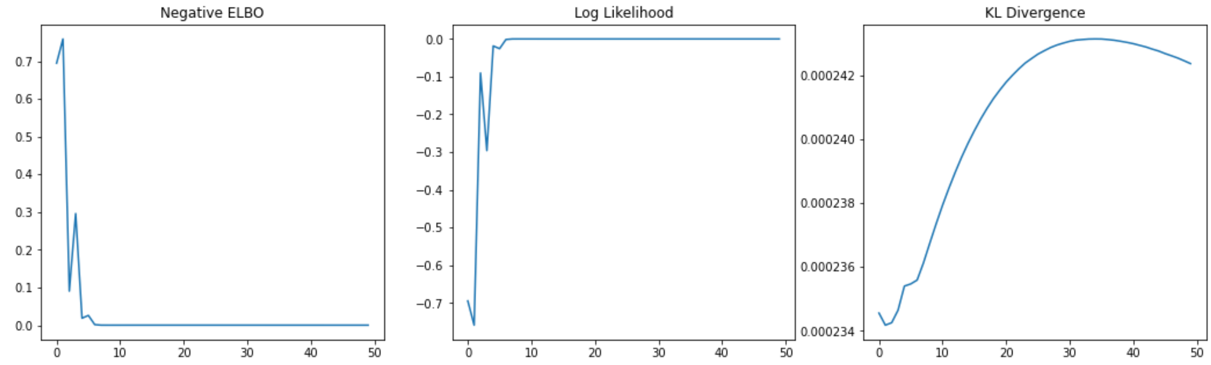
```
    c = A[0,:] - A[1,:]
    d = b[0] - b[1]
    # then the decision boundary is c[0] * x_0 + c[1] * x_1 + d = 0, i.e., np.dot(c,
    # general solution for higher dimension data could be found by scipy.linalg.null
    # but hard to visualize anyway

    return lambda x_1: -1.0 * (d+c[1]*x_1) / c[0]
```

In [17]:
```
sto_model = StoLogisticRegression(in_features=2, out_features=2, bias=True, depth=16)
optimizer = optim.Adam(sto_model.parameters(), lr=0.03)
loss_list, ll_list, kl_list = [], [], []

fig, ax = plt.subplots(1, figsize=(6,6))

for epoch in range(50):
    log_probs, log_det_jacobian = sto_model(X_train)
    loss, ll, kl = sto_model.calc_loss(log_probs, y_train, log_det_jacobian)
    loss_list.append(loss.item())
    ll_list.append(ll.item())
    kl_list.append(kl.item())
    print("Epoch {} Loss {} Neg Log Likelihood {} KL (scaled) {}".format(epoch,
        loss.item(), -ll.item(), kl.item()))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ax.clear()
    for this_y, color in zip(y_unique, colors):
        this_X = X_train[y_train == this_y]
        ax.scatter(this_X[:, 0], this_X[:, 1], s=15,
                   c=color[np.newaxis, :],
                   alpha=0.5, edgecolor='k',
                   label="Class %s" % this_y)

    fn = sto_model.get_decision_boundary()
    x1_s = torch.arange(-6, 6, 0.1)
    x0_s = fn(x1_s)
    ax.plot(x0_s, x1_s)
    ax.set_xlim(-8,8)
    ax.set_ylim(-8,8)
    ax.set_title("Epoch {}".format(epoch))

    display.clear_output(wait=True)
    display.display(fig)
    plt.pause(0.005)
```



The loss curve has a sudden jump in early stages, what's the reason and how to explain?

In [18]:
```
plt.figure(figsize=(18,5))
plt.subplot(1,3,1)
plt.plot(loss_list)
plt.title("Negative ELBO")
plt.subplot(1,3,2)
plt.plot(ll_list)
plt.title("Log Likelihood")
plt.subplot(1,3,3)
plt.plot(kl_list)
plt.title("KL Divergence")
plt.show()
```



In [19]:
```
mean_prob, variances = sto_model.make_prediction(X_test)
acc = classification_accuracy(mean_prob, y_test)
print("Classification Accuracy is {}".format(acc))

# the variance of the predicted class
_, pred_label = torch.max(mean_prob, dim=1)
pred_var = variances[torch.arange(y_test.numel()), pred_label]
```

```
Classification Accuracy is 1.0
```

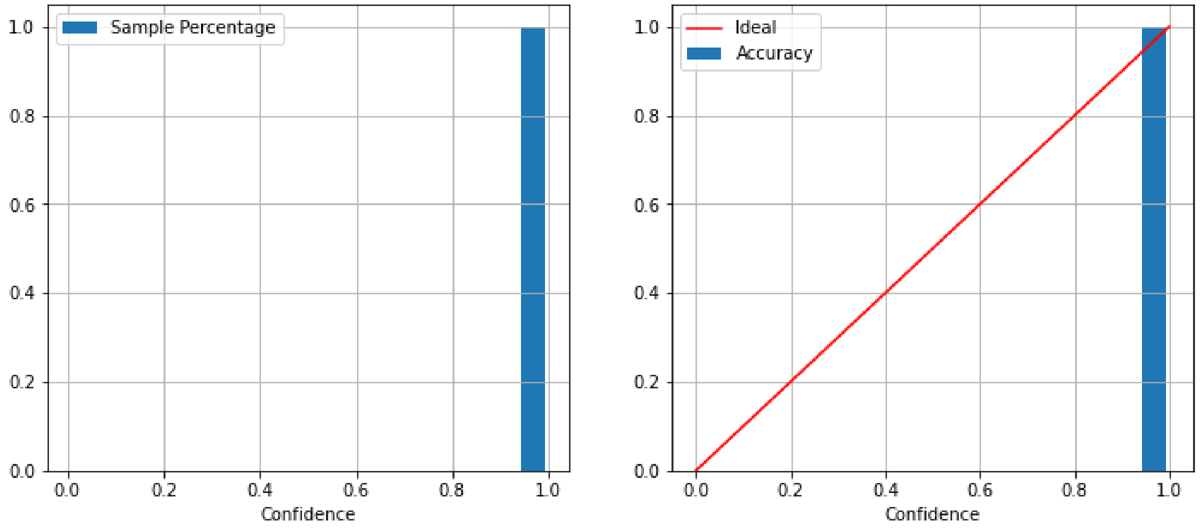
Tried like 20 times, everytime the stochastic model has nicer ece score/plot than the deterministic one

the deterministic model sometimes got a bad initialization, and the result will be bad

we can say the stochastic model is more stable

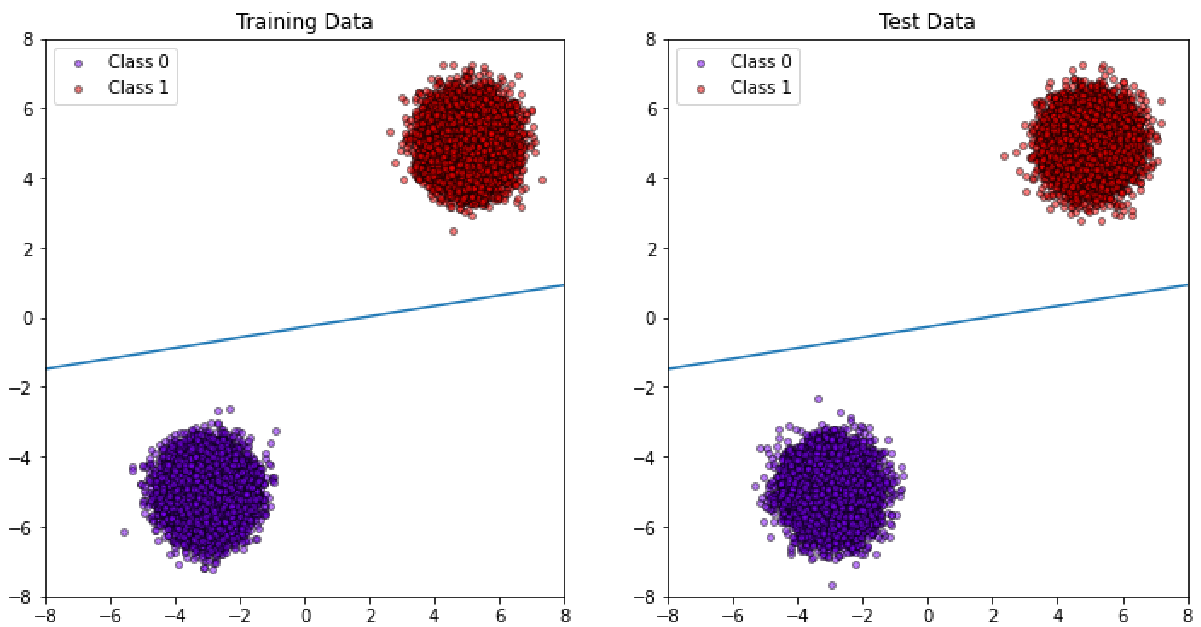
In [20]:
```
ece_loss = ECELoss(n_bins=15)
# the model outputs log softmax result, so use exp to get back probability
ece_score = ece_loss(mean_prob, y_test, is_logit=False)
print(ece_score)
ece_loss.plot_acc_conf_gap()
```

```
tensor([4.2081e-05], grad_fn=<AddBackward0>)
```



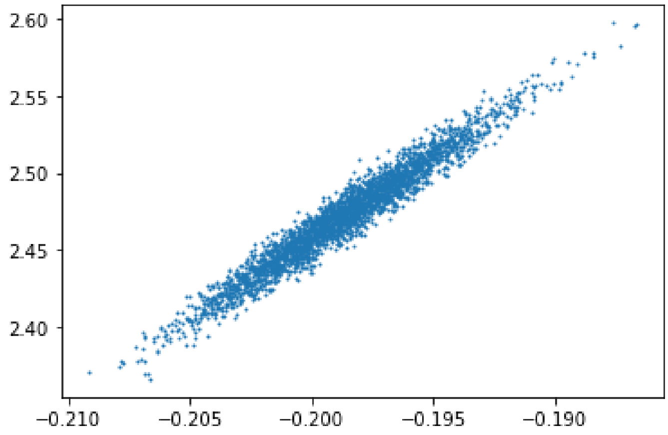
In [21]:

```python
y_unique = np.unique(y)
colors = cm.rainbow(np.linspace(0.0, 1.0, y_unique.size))

plt.figure(figsize=(12, 6))
plt.subplot(1,2,1)
for this_y, color in zip(y_unique, colors):
    this_X = X_train[y_train == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=15,
                c=color[np.newaxis, :],
                alpha=0.5, edgecolor='k',
                label="Class %s" % this_y)
# the decision bounday is not necessarily the same everytime
# but it always passes the origin
fn = sto_model.get_decision_boundary()
x1_s = torch.arange(-6, 6, 0.1)
x0_s = fn(x1_s)
plt.plot(x0_s, x1_s)

plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.legend(loc="best")
plt.title("Training Data");

plt.subplot(1,2,2)
for this_y, color in zip(y_unique, colors):
    this_X = X_test[y_test == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=15,
                c=color[np.newaxis, :],
                alpha=0.5, edgecolor='k',
                label="Class %s" % this_y)
# the decision bounday is not necessarily the same everytime
# but it always passes the origin
fn = sto_model.get_decision_boundary()
x1_s = torch.arange(-6, 6, 0.1)
x0_s = fn(x1_s)
plt.plot(x0_s, x1_s)

plt.xlim(-8, 8)
plt.ylim(-8, 8)
plt.legend(loc="best")
plt.title("Test Data");

plt.show()
```



Generate samples from the base distribution, and then use the flow to transform them

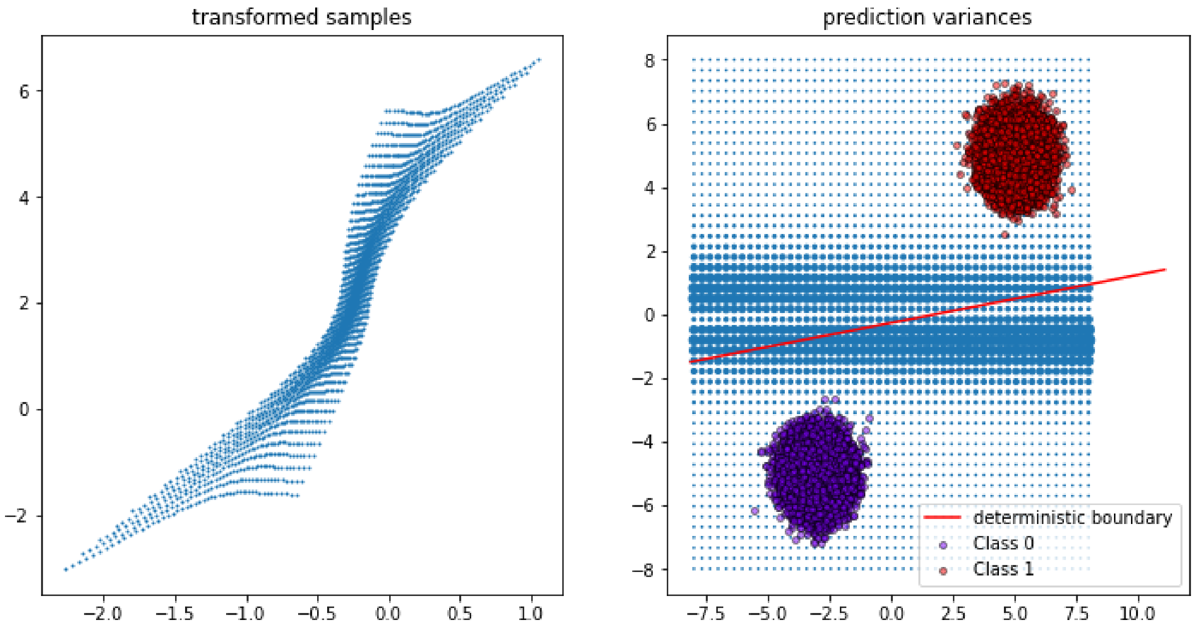
In [22]:
```
with torch.no_grad():
    original = sto_model.base_dist.sample((3000, 2))
    transformed, _ = sto_model.norm_flow(original)
plt.scatter(transformed[:,0], transformed[:,1], s=1);
```



Create a grid of (x, y) pairs, put them through the flow to see how the flow transforms the space. Then put them through the whole network to see how the variances distributes across the 2d space.

In [23]:
```
xs = torch.linspace(-8, 8, 50)
ys = torch.linspace(-8, 8, 50)
grid_x, grid_y = torch.meshgrid(xs, ys)
xy_pairs = torch.cat((grid_x.unsqueeze(-1), grid_y.unsqueeze(-1)), dim=-1)
xy_pairs = xy_pairs.reshape(-1,2)

transformed, _ = sto_model.norm_flow(xy_pairs)
transformed = transformed.detach().numpy()
mean_prob, variances = sto_model.make_prediction(xy_pairs)
variances = torch.clamp(variances, min=torch.max(variances).detach().numpy()/1e4)
_, pred_label = torch.max(mean_prob, dim=1)
var_pred = variances[torch.arange(xy_pairs.shape[0]), pred_label]
```

In [27]:
```
plt.figure(figsize=(12, 6))
var_vis = torch.sqrt(var_pred/(var_pred.min()+np.finfo(float).eps)).detach().numpy()

plt.subplot(1,2,1)
plt.scatter(transformed[:, 0], transformed[:,1], s=1);
plt.title("transformed samples")

plt.subplot(1,2,2)
plt.scatter(xy_pairs[:,0].detach(), xy_pairs[:,1].detach(), s=var_vis*0.5)
for this_y, color in zip(y_unique, colors):
    this_X = X_train[y_train == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=15,
                c=color[np.newaxis, :],
                alpha=0.5, edgecolor='k',
                label="Class %s" % this_y)

fn = sto_model.get_decision_boundary()
x1_s = torch.arange(-1.5, 1.5, 0.1)
x0_s = fn(x1_s)
plt.plot(x0_s, x1_s, color="r", label="deterministic boundary")
plt.legend(loc="best")
plt.title("prediction variances");
```



Generally, both the deterministic model and stochastic model can correctly classify this simple dataset, but the stochastic model has better ECE score, and thus it's bette calibrated.

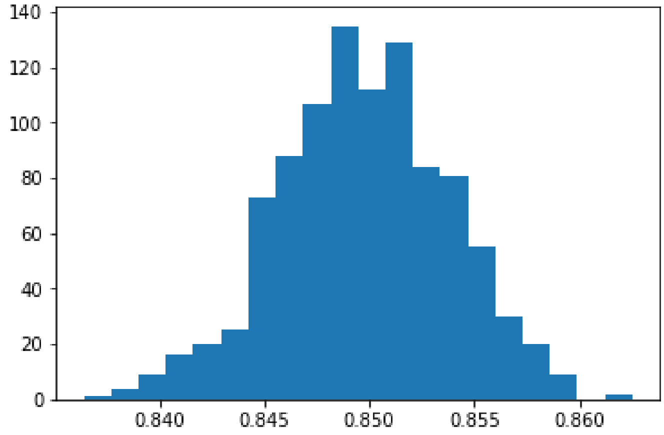
The left figure above shows how the flow transforms the sample space, it is obtained by generating a grid, and put the grid points into the flow. glad to see the flow has learned something ....

The right figure above shows the predictive variances of the grid points, and it's obtained by putting the grid points through the whole model (not just the flow). In the stochastic model, sometimes the deterministic boundary doesn't right, but the flow part could make the model better. A possible reason is that I trained those parameters all together. The deterministic part will lose gradient once the flow has learend to correctly transform the inputs, since the accuracy will remain very high and stable (the flow handles nearly everything).

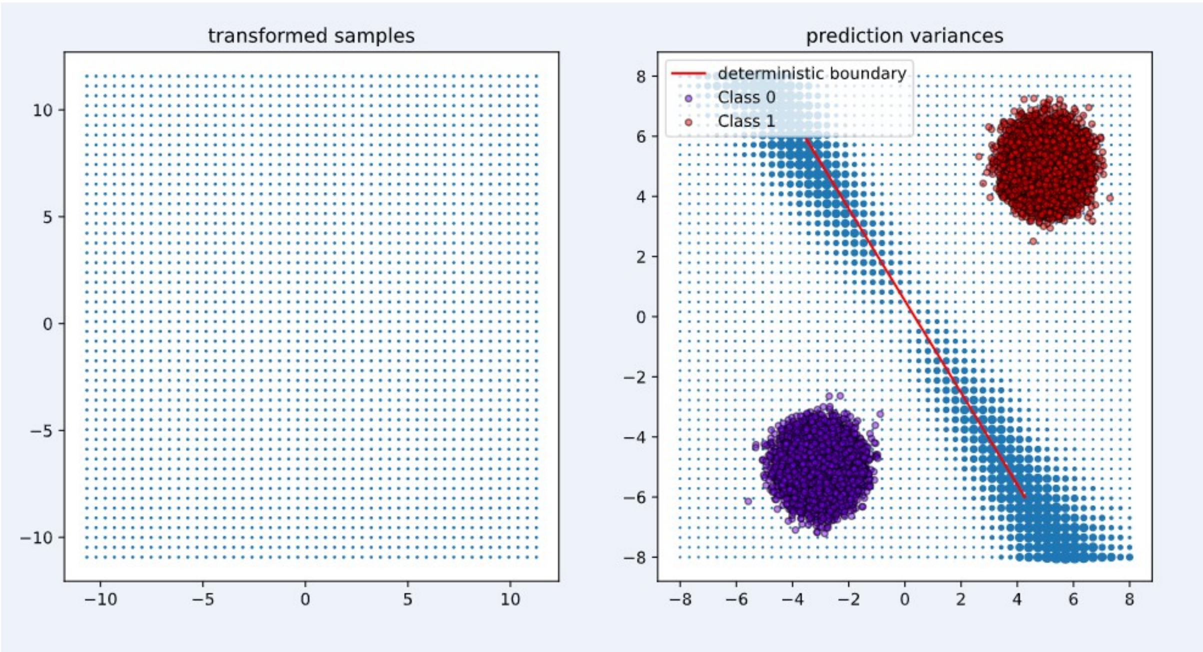
The size of the points shows the variances in predictive result (larger points have higher variances), and we can see a clear boundary in the middle, while the rest of intermediate zone has higher variance.

if we pick a point with high variance (7.5, -1), and see the actual predictive distribution for this sigle point

In [28]:
```
mean_prob, variances, all_probs = sto_model.make_prediction(torch.tensor([7.5, -1]).resh
class_label = torch.argmax(mean_prob)
label_prob = all_probs.squeeze()[:, class_label].detach().numpy()
plt.hist(label_prob, 20);
plt.show()
```



If we remove the planar flows by setting the flow depth to 0 (only the affine transform remains), the picture looks like the flowing, and the non-linearity in the flow part just disappeared.



In [ ]: